Design With Verilog
Dr. Paul D. Franzon

Outline
1. Procedural Examples
2. Continuous Assignment
3. Structural Verilog
5. Common Problems
6. More sophisticated examples

References
1. Quick Reference Guides
2. Smith & Franzon, Chapter 2-6, 8, Appendix A

Always Design Before Coding
Lexical Conventions in Verilog

Logic values:

<table>
<thead>
<tr>
<th>Logic Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Zero, low, or false</td>
</tr>
<tr>
<td>1</td>
<td>One, high or true</td>
</tr>
<tr>
<td>Z or z or ?</td>
<td>High impedance, tri-stated or floating</td>
</tr>
<tr>
<td>X or x</td>
<td>Unknown, uninitialized, or Don’t Care</td>
</tr>
</tbody>
</table>

Integers:

```
1'b1; 4'b0;
size 'base value; size = # bits, HERE: base = binary
```

Other bases:  h = hexadecimal,  d = decimal (which is the default)

Examples:  10  3'b1
          8'hF0  8'hF
          5'd11  2'b10
Procedural Blocks

Code of the type
always@(input1 or input2 or ...)
begin
    if-then-else or case statement
end

is referred to as Procedural Code because the statements between the `begin` and the `end` are executed procedurally, or in order.

- Note that all variables assigned (i.e. on the left hand side) in procedural code must be of a register data type. Here type `reg` is used.
  - Because a variable is of type `reg` does NOT mean it is a register or flip-flop.
- The procedural block is executed when triggered by the `always@` statement.
  - Then the statements that follow are executed once
  - The statements in parentheses `( ... )` are referred to as the sensitivity list.
  - `always@(posedge clock)` executes whenever a positive edge on the clock occurs
  - `always@(value)` executes whenever any of the statements in parentheses change
**Blocking vs. Non-Blocking Assignment**

What is the difference between the following code segments?

```verbatim
initial
begin
    a = 4'h3; b = 4'h4;
end

always@(posedge clock)
begin
    c = a + b;
    d = c + a;
end;
```

* = is referred to as a blocking assignment, because execution of subsequent code is blocked until this statement completes. Essentially, $c = a + b$; and $d = c + a$; are performed in series.

* <= is referred to as non-blocking assignment. Essentially, $c <= a + b$; and $d <= c + a$; are performed in parallel.

**Always Use <= when building registers to prevent possible races.**
Registers

Some Flip Flop Types:

```verilog
reg Q0, Q1, Q2, Q3, Q4;
// D Flip Flop
always@(posedge clock)
    Q0 <= D;
// D Flip Flop with asynchronous reset
always@(posedge clock or negedge reset)
    if (!reset) Q1 <= 0;
    else Q1 <= D;
// D Flip Flop with synchronous reset
always@(posedge clock)
    if (!reset) Q2 <= 0;
    else Q2 <= D;
// D Flip Flop with enable
always@(posedge clock)
    if (enable) Q3 <= D;
// D Flip Flop with synchronous clear and preset
always@(posedge clock)
    if (!clear) Q4 <= 0;
    else if (!preset) Q4 <= 1;
    else Q4 <= D;
```

Note:
Registers with asynchronous reset are smaller than those with synchronous reset
+ don’t need clock to reset
BUT it is a good idea to synchronize reset at the block level to reduce impact of noise.
Behavior ➔ Function

What do the following code fragments synthesize to?

```verilog
reg foo;
always @(a or b or c)
begin
  if (a)
    foo = b | c;
  else foo = b ^ c;
end

reg (foo)
always@(clock or a)
  if (clock)
    foo = a;
end
```

Each variable has to be assigned every time the code is executed, so as not to specify memory, or an unintended latch.
Sketch the logic being described:

```vhdl
input [1:0] sel;
input [3:0] A;
reg Y;
always@(sel or A)
casex (sel)
  0 : Y = A[0];
  1 : Y = A[1];
  2 : Y = A[2];
  3 : Y = A[3];
  default : Y = 1’bx;
endcase
```

The 'default' here is not essential but useful to
(1) Enforce discipline preventing latches
(2) Ensures proper startup behavior (see later)
Synopsys optimizes it out in this case.

Case statement:
Alternatives evaluated in sequence.
ONLY first correct match used.
Behavior ➔ Function

Sketch the truth table, and describe the logic:

```plaintext
input [3:0] A;
reg [1:0] Y;
always@(A)
  casex (A)
    8'b0001 : Y = 0;
    8'b0010 : Y = 1;
    8'b0100 : Y = 2;
    8'b1000 : Y = 3;
    default : Y = 2'bx;
  endcase
```

4-2 encoder
Behavior \rightarrow Function

Sketch the truth table, and describe the logic:

```verilog
input [3:0] A;
reg [1:0] Y;
always@(A)
case (A)
  4'b1xxx : Y = 0;
  4'b01xx : Y = 1;
  4'b001x : Y = 2;
  4'b0000 : Y = 3;
  4'b0001 : Y = 0;
  default : Y = 2'bx;
endcase
```
Sketch the logic:

```vhdl
input [2:0] A;
reg [7:0] Y;
always@(A or B or C)
begin
    Y = B + C;
case x (A)
        8'b1xx : Y = B - C;
        8'b000 : Y = B | C;
        8'b001 : Y = B & C;
    endcase
end
```

**Behavior ➔ Function**
Behavior ➔ Function

integer i, N;
parameter N=7;
reg [N:0] A;
always@(A)
begin
    OddParity = 1'b0;
    for (i=0; i<=N; i=i+1)
        if (A[i]) OddParity = ¬OddParity;
end
**Procedural Code**

- `always@(posedge clock) results in what?`

- Variables assigned procedurally are declared as what type?

- What type of assignment should be used when specifying flip-flops?

- When is the block evaluated?
Exercises

Implement a 2-bit Grey scale encoder:
(I.e. Binary encoding of 1..4 differ
by only 1 bit)

Implement hardware that counts the # of 1’s in input [7:0] A;
Operators
Continuous Assignment

Sketch the logic being specified ...

```verbatim
input [3:0] A, B;
wire [3:0] C, E;
wire D, F, G;
assign C = A ^ B;
assign D = ~A;
assign F = A[0] ? B[0] : B[1];
assign G = (A == B);
```
Continuous Assignment

Sketch the logic being specified …

```vhdl
input A, B, C;
tri F;
assign F = A ? B : 1'bz;
assign F = ~A ? C : 1'bz;
```
Sketch the logic being specified …

```verbatim
input [3:0] A, B, C;
wire [3:0] F, G;
wire H;

assign F = A + B + C + D;
assign G = (A+B) + (C+D);
assign H = C[A[1:0]];
```
Continuous Assignment

- When are expressions evaluated?

- What types of variables can be assigned?

- Is this the only way to build synthesizable tri-state buffers?

Exercise -- Use Continuous Assignment to Make an even Parity Generator:

```verilog
wire [31:0] A;
wire even_parity;
```

assign even_parity = ~^A;
**Structural Verilog**

Complex modules can be put together by ‘building’ (instancing) a number of smaller modules.

e.g. Given the 1-bit adder module with module definition as follows, build a 4-bit adder with carry_in and carry_out

```
module OneBitAdder (CarryIn, In1, In2, Sum, CarryOut);
```

4-bit adder:

```
module FourBitAdder (Cin, A, B, Result, Cout);

input        Cin;
input  [3:0] A, B;
output [3:0] Result;
output       Cout;
wire [3:1] chain;

OneBitAdder u1 (.CarryIn(Cin), .In1(A[0]), .In2(B[0]), .Sum(Result[0]), .CarryOut(chain[1]));
OneBitAdder u2 (.CarryIn(chain[1]), .In1(A[1]), .In2(B[1]), .Sum(Result[1]), .CarryOut(chain[2]));
OneBitAdder u4 (Chain[3], A[3], B[3], Result[3], Cout); // in correct order
endmodule
```
Structural Example

- Sketch:
**Structural Verilog**

Features:

Four copies of the same module (OneBitAdder) are built (‘instanced’) each with a unique name (u1, u2, u3, u4).

Module instance syntax:

```
OneBitAdder u1 (.CarryIn(Cin),
Module Name          Instance Name      Port Name inside Module (optional)
```

All nets connecting to outputs of modules must be of wire type (wire or tri):

```
wire [3:1] chain;
```
Applications of Structural Verilog

- To Assemble modules together in a hierarchical design.
- Final gate set written out in this format (“netlist”).
- Design has to be implemented as a module in order to integrate with the test fixture
module counter (clock, in, latch, dec, zero);
  input [3:0] in;
  input clock, latch, dec;
  output zero;
  wire [value3], [value1], [value53[2]], [value53[0]], \n[54[0]], [value2], [value0], [value53[1]], [value53[3]], n103, n104, n105, n106, n107, n108, n109, n110, n111, n112, n113, n114, n115;
  NOR2 U36 (.Y(n107), .A0(n109), .A1(value[2]));
  NAND2 U37 (.Y(n109), .A0(n105), .A1(n103));
  NAND2 U38 (.Y(n114), .A0(value[1]), .A1(value[0]));
  NOR2 U39 (.Y(n115), .A0(value[3]), .A1(value[2]));
  XOR2 U40 (.Y(n110), .A0(value[2]), .A1(n109));
  NAND2 U41 (.Y(n113), .A0(n109), .A1(n114));
  INV U42 (.Y(n54[0]), .A(n106));
  INV U43 (.Y(n108), .A(n109));
  AOI21 U44 (.Y(n106), .A0(n112), .A1(dec), .B0(latch));
  INV U45 (.Y(zero), .A(n112));
  NAND2 U46 (.Y(n112), .A0(n110), .A1(n108));
  OAII1 U47 (.Y(n111), .A0(n107), .A1(n104), .B0(n112));
  DSEL2 U48 (.Y(value53[3]), .D0(n111), .D1(in[3]), .SO(latch));
  DSEL2 U49 (.Y(value53[2]), .D0(n110), .D1(in[2]), .SO(latch));
  DSEL2 U50 (.Y(value53[1]), .D0(n113), .D1(in[1]), .SO(latch));
  DSEL2 U51 (.Y(value53[0]), .D0(n103), .D1(in[0]), .SO(latch));
  EDFF \value_reg[3] (.Q(value[3]), .QBAR(n104), .CP(clock), .D(value53[3]), .E(n54[0]));
  EDFF \value_reg[2] (.Q(value[2]), .CP(clock), .D(value53[2]), .E(n54[0]));
  EDFF \value_reg[1] (.Q(value[1]), .QBAR(n103), .CP(clock), .D(value53[1]), .E(n54[0]));
  EDFF \value_reg[0] (.Q(value[0]), .QBAR(n105), .CP(clock), .D(value53[0]), .E(n54[0]));
endmodule
Common Problems and Fixes

Unintentional Latches

- How to detect: Found by Synopsys after “read” command
- How to fix: Make sure every variable is assigned for every way code is executed
- What happens if unfixed: Glitches on “irregular clock” to latch cause set up and hold problems in actual hardware (transient failures)

Problem Code: Possible Fix:

```verbatim
global always@(A or B)
    begin
        if (A) C = ~B;
        else D = |B;
    end
```
Common Problems and Fixes

Incomplete Sensitivity List
- How to detect: After “read” command synopsys says “Incomplete timing specification list”
- How to fix: All module inputs have to appear in sensitivity list
- What happens if unfixed: Since simulation results won’t match what actual hardware will do, bugs can remain undetected

<table>
<thead>
<tr>
<th>Problem Code</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>always @(A or B) begin</td>
<td>fix</td>
</tr>
<tr>
<td>if (A) C = B ^ A;</td>
<td>else C = D &amp; E;</td>
</tr>
<tr>
<td>F = C</td>
<td>A;</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

Note: C not in list
Common Problems and Fixes

Unintentional Wired-OR logic
- How to detect: After “read” command synopsys says “variable assigned in more than one block”
- How to fix: Redesign hardware so that every signal is driven by only one piece of logic (or redesign as a tri-state bus if that is the intention)
- What happens if unfixed: Unsynthesizable.

This is a symptom of NOT designing before coding

<table>
<thead>
<tr>
<th>Problem Code</th>
<th>Possible Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>always@(A or B)</td>
<td></td>
</tr>
<tr>
<td>if (A) C =</td>
<td>B;</td>
</tr>
<tr>
<td>always@(D or E)</td>
<td></td>
</tr>
<tr>
<td>if (D) C = ^E;</td>
<td></td>
</tr>
</tbody>
</table>
Common Problems and Fixes

Improper Startup

- How to detect: Can’t
- How to fix: Make sure “don’t cares” are propagated
- What happens if unfixed: Possible undetected bug in reset logic

<table>
<thead>
<tr>
<th>Problem Code</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td>always@(posedge clock)</td>
<td>if (A) Q &lt;= D;</td>
</tr>
<tr>
<td>always@(Q or E)</td>
<td>case (Q)</td>
</tr>
<tr>
<td></td>
<td>0 : F = E;</td>
</tr>
<tr>
<td></td>
<td>default : F = 1;</td>
</tr>
<tr>
<td></td>
<td>endcase</td>
</tr>
<tr>
<td>clock : 00001111</td>
<td>D : xxxxxxxx</td>
</tr>
<tr>
<td></td>
<td>E : 11110000</td>
</tr>
<tr>
<td></td>
<td>Q :</td>
</tr>
<tr>
<td></td>
<td>F :</td>
</tr>
</tbody>
</table>

©2003, Dr. Paul D. Franzon, www.ece.ncsu.edu/erl/faculty/paulf.html
Debug

- How to prevent a lot of need to debug:
  - Carefully think through design before coding
  - Simulate “in your head”

- How to debug:
  - Track bug point back in design and back in time
    - Check if each “feeding” signal makes sense
  - Compare against a “simulation in your head”
  - If all else fails, recode using a different technique
Larger Examples: Linear Feedback Shift Register (fn)

```verilog
module LFSR_FN (Clock, Reset, Y1, Y2);
  input Clock, Reset;
  output [7:0] Y1, Y2;
  reg [7:0] Y1, Y2;
  parameter [7:0] seed1 = 8'b01010101;
  parameter [7:0] seed2 = 8'b01110111;

  function [7:0] LFSR_TAPS8_FN;
  input [7:0] A;
  integer N;
  parameter [7:0] Taps = 8'b10001110;
  reg Bits0_6_Zero, Feedback;
  begin
    Bits0_6_Zero = ~| A[6:0];
    for (N=7; N>=1; N=N-1)
      else LFSR_TAPS8_FN[N] = A[N-1]; LFSR_TAPS8_FN[0] = Feedback;
  end
endfunction /* LFSR_TAPS8_FN */

  /* Build 2 LFSRs using the LFSR_TAPS8_TASK */
  always@(posedge Clock or negedge Reset)
    if (!Reset) Y1 <= seed1; else Y1 <= LFSR_TAPS8_FN (Y1);
  always@(posedge Clock or negedge Reset)
    if (!Reset) Y2 <= seed2; else Y2 <= LFSR_TAPS8_FN (Y2);
endmodule
```
LFSR

- Sketch Design
ECE 520 Class Notes

LFSR (Task)

module LFSR_TASK (clock, Reset, Y1, Y2);
input clock, Reset;
output [7:0] Y1;
reg [7:0] Y1;
parameter [7:0] seed1 = 8'b01010101; parameter [7:0] Taps1 = 8'b10001110;

task LFSR_TAPS8_TASK;
input [7:0] A; input [7:0] Taps; output [7:0] Next_LFSR_Reg;
integer N; reg Bits0_6_Zero, Feedback; reg [7:0] Next_LFSR_Reg;
begin
    for (N=7; N>=1; N=N-1)
        else Next_LFSR_Reg[N] = A[N-1];
    Next_LFSR_Reg[0] = Feedback;
end
endtask /* LFSR_TAP8_TASK */

always@(posedge clock or negedge Reset)
    if (!Reset) Y1 = seed1;
    else LFSR_TAPS8_TASK (Y1, Taps1, Y1);
endmodule
module RegFile (clock, WE, WriteAddress, ReadAddress, WriteBus, ReadBus);
input  clock, WE;
input  [4:0] WriteAddress, ReadAddress;
input  [15:0] WriteBus;
output [15:0] ReadBus;
reg [15:0]   Register [0:31];   // thirty-two 16-bit registers

// provide one write enable line per register
wire [31:0] WElines;
integer i;

// Write '1' into write enable line for selected register
assign WElines = (WE << WriteAddress);

always@(posedge clock)
  for (i=0; i<=31; i=i+1)
    if (WElines[i]) Register[i] <= WriteBus;

assign ReadBus  =  Register[ReadAddress];
endmodule
Register File

- Sketch Design

Alternative:
```verilog
always@(posedge clock)
if (WE) Register[WriteAddress] = WriteBus;
```
tends to result in more logic, with its implied address decoder.
**Sample Problem**

- **Accumulator:**
  - Design an 8-bit adder accumulator with the following properties:
  - While ‘accumulate’ is high, adds the input, ‘in1’ to the current accumulated total and add the result to the contents of register with output ‘accum_out’.
  - When ‘start’ is high (‘accumulate’ will be low) feed ‘in1’ into accumulator
  - The ‘overflow’ flag O/P goes high if the adder overflows (unsigned numbers) (don’t register)

```verilog
Module accumulator (accumulate, clear, in1, accum_out, overflow);

input accumulate, clear;
input [7:0] in1;
output [7:0] accum_out;
output overflow;

reg [7:0] accum_out;
wire [7:0] accum_in;
wire carry, overflow, enable;

Fourbitadder ul (1'b0, in1[3:0], accum_out[3:0], result[3:0], carry);
Fourbitadder ul (carry, in1[7:4], accum_out[7:4], result[3:0], overflow);

assign enable = accumulate || clear;
assign accum_in = clear ? 8'b0 : result;
always@(posedge clock)
  if (enable) accum_out <= accum_in;
endmodule
```
Verilog

- Hint: \( \{Cout, sum\} = A + B; \)
  - VL recognizes that adding \( n\)-bit numbers gives an \( n+1 \) bit result

```verilog
module Accumulator (clock, start, accumulate, in1, accum_out, overflow);
  input clock, start, accumulate;
  input [7:0] In1;
  output [7:0] accum_out;
  output overflow;
  reg [7:0] next_accum_out, accum_out;
  reg overflow;
  always@(posedge clock)
    accum_out <= next_accum_out;
  always@(start or accumulate or in1 or accum_out)
    begin
      overflow = 1'b0;
      if (start) next_accum_out = in1;
      else if (accumulate)
        {overflow, next_accum_out} = accum_out + in1;
      else next_accum_out = accum_out;
    end
endmodule
```
Summary

- What are the two HDL designers “mantra’s” so far?

- What are the three basic VL constructs?

- What is structural VL used for?

- What are 3 common problems?